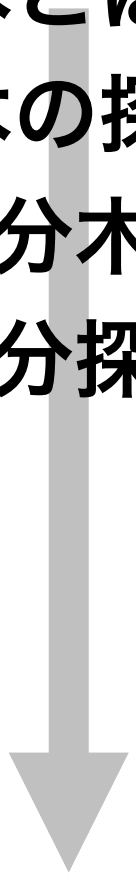
木 (1)

クラスライブラリ応用

斉藤（裕）

本日の内容

木とは
木の探索
2分木
2分探索木



本日の内容

木とは

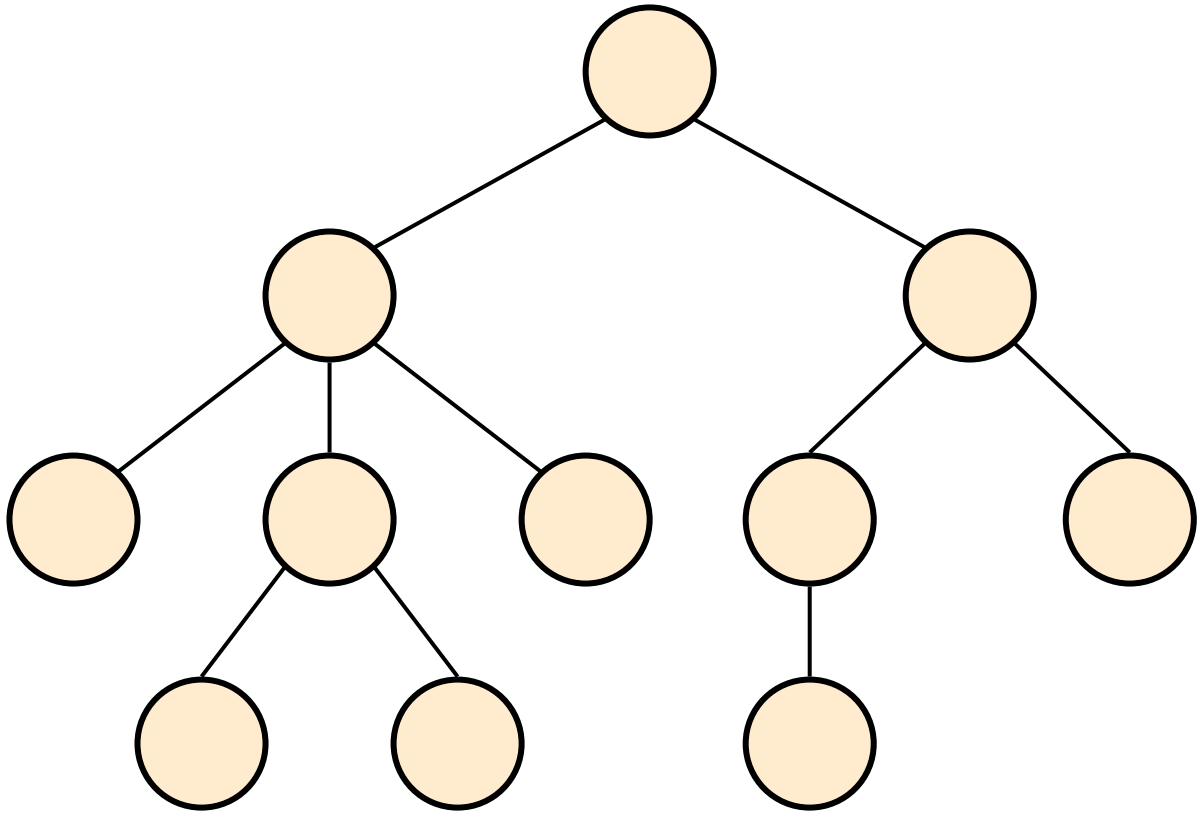
木の探索

2分木

2分探索木

木

・階層的な関係を表現するデータ構造

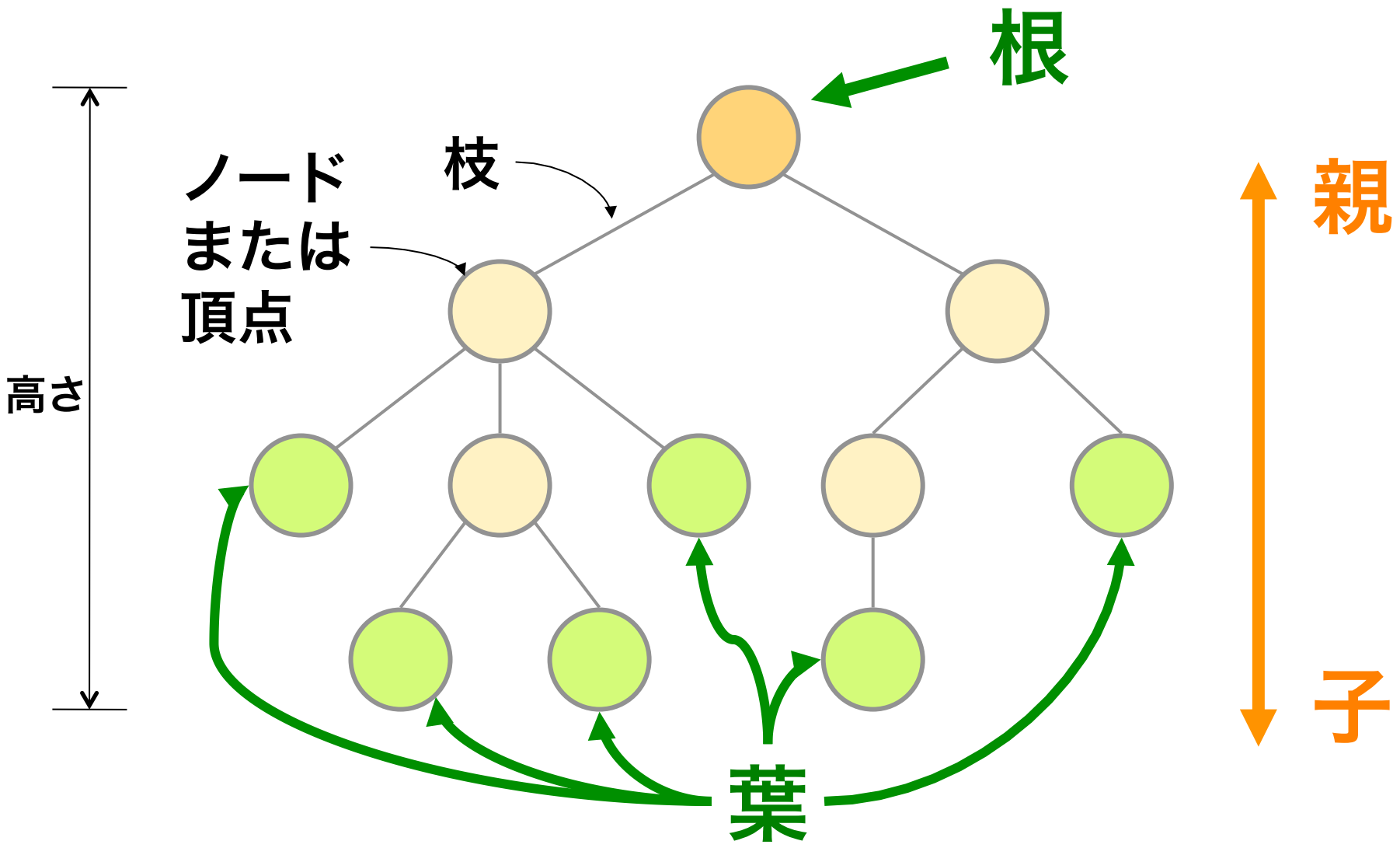


現実世界での木

- トーナメント戦の対戦順
- 家系図
- 住所の表現形式
- インターネットのアドレス
  - 例: www.im.dendai.ac.jp

人間の思考方法は
全体を部分に分けて理解しようとする特性がある。

用語集



本日の内容

木とは

木の探索

2分木

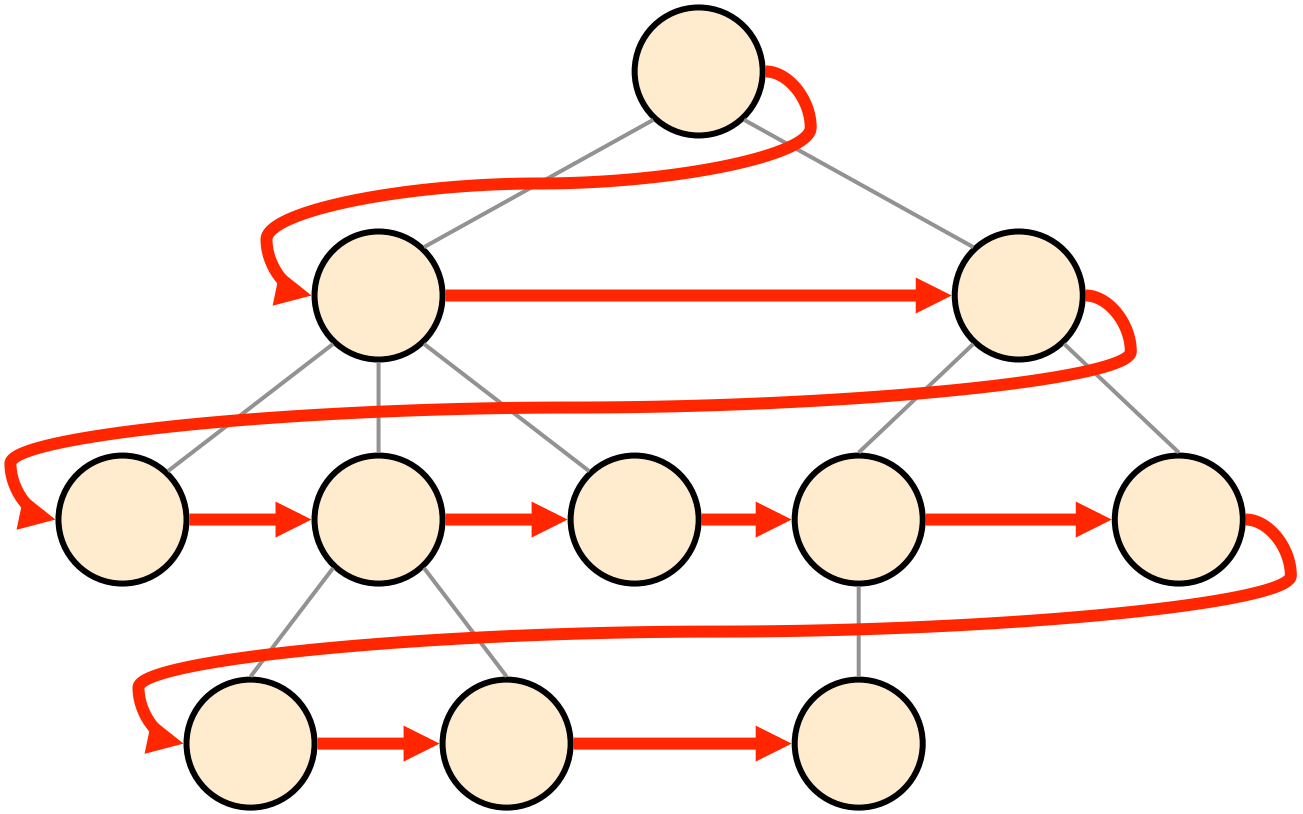
2分探索木

木の探索

- 木のノードを順番にたどる方法
- 「走査」とも言う
- 2とおりの方法
  - **幅優先探索**
  - **深さ優先探索**

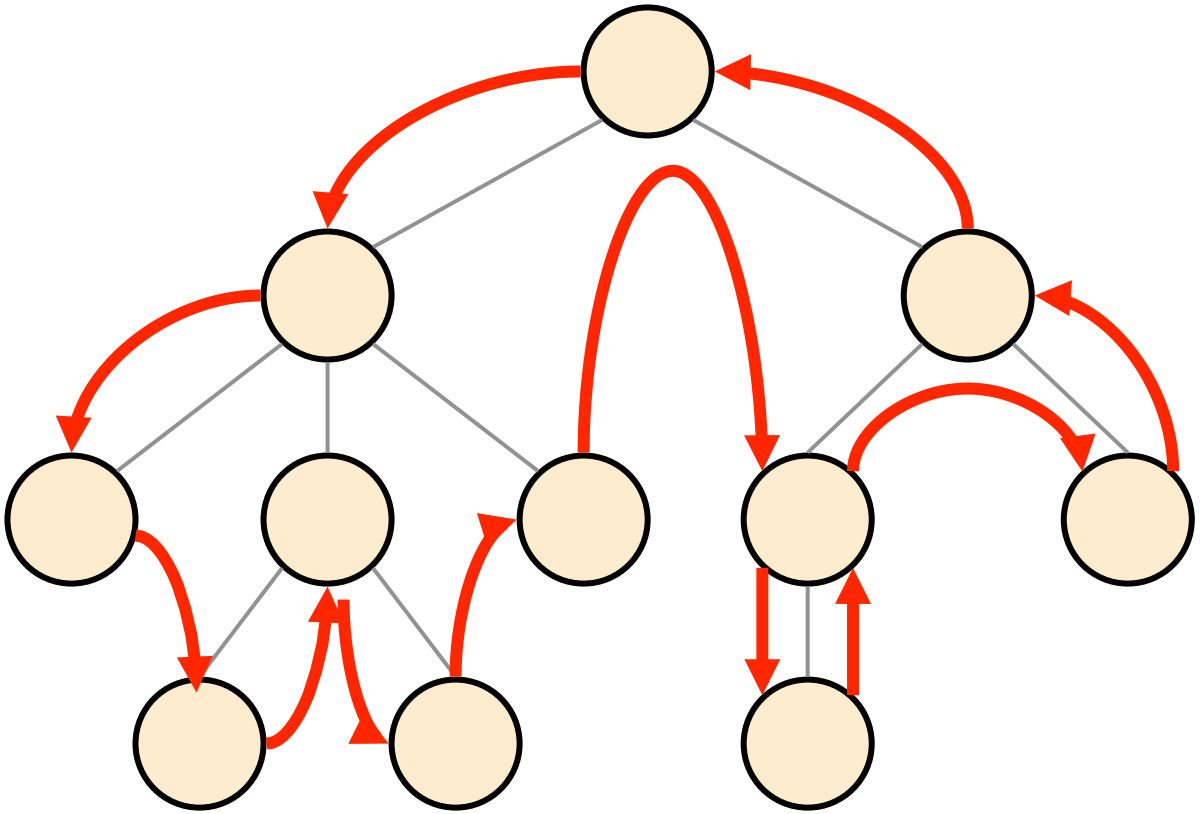
幅優先探索

左から右になぞり、子の階層に進む



深さ優先探索

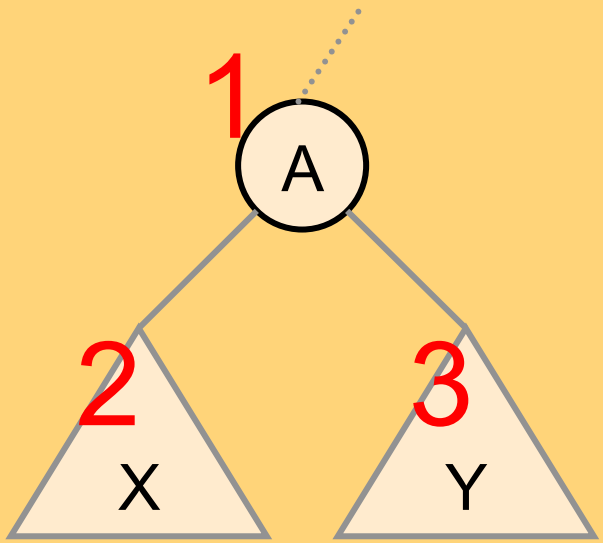
行き止まりになるまで下る
進めなくなったら引き返して別の道を選ぶ



深さ優先探索

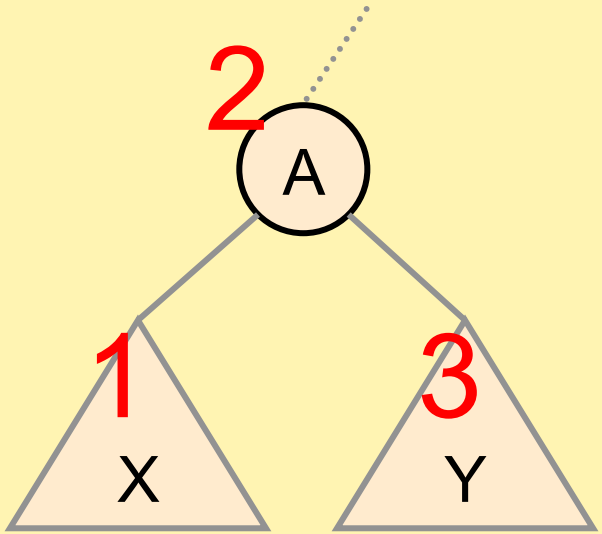
前順走査（行きがけ順）

- preorderとも言う
- 自分のノードを調べてから、子を調べる
- A→X→Y



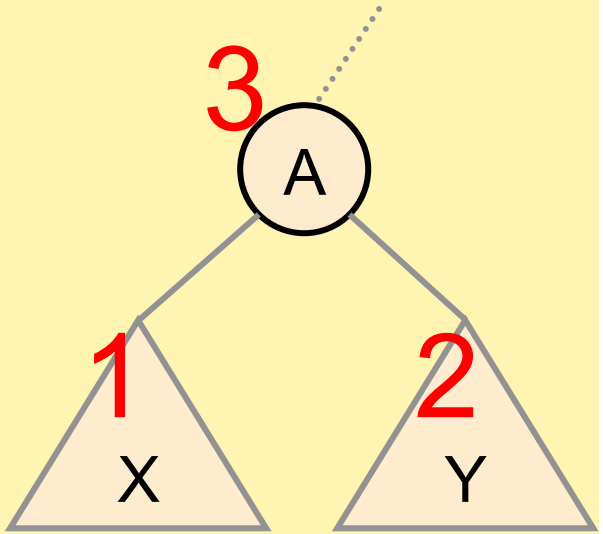
間順走査（通りがけ順）

- inorderとも言う
- 一方の子を調べてから、自分のノードを調べ、次にもう一方の子を調べる
- X→A→Y



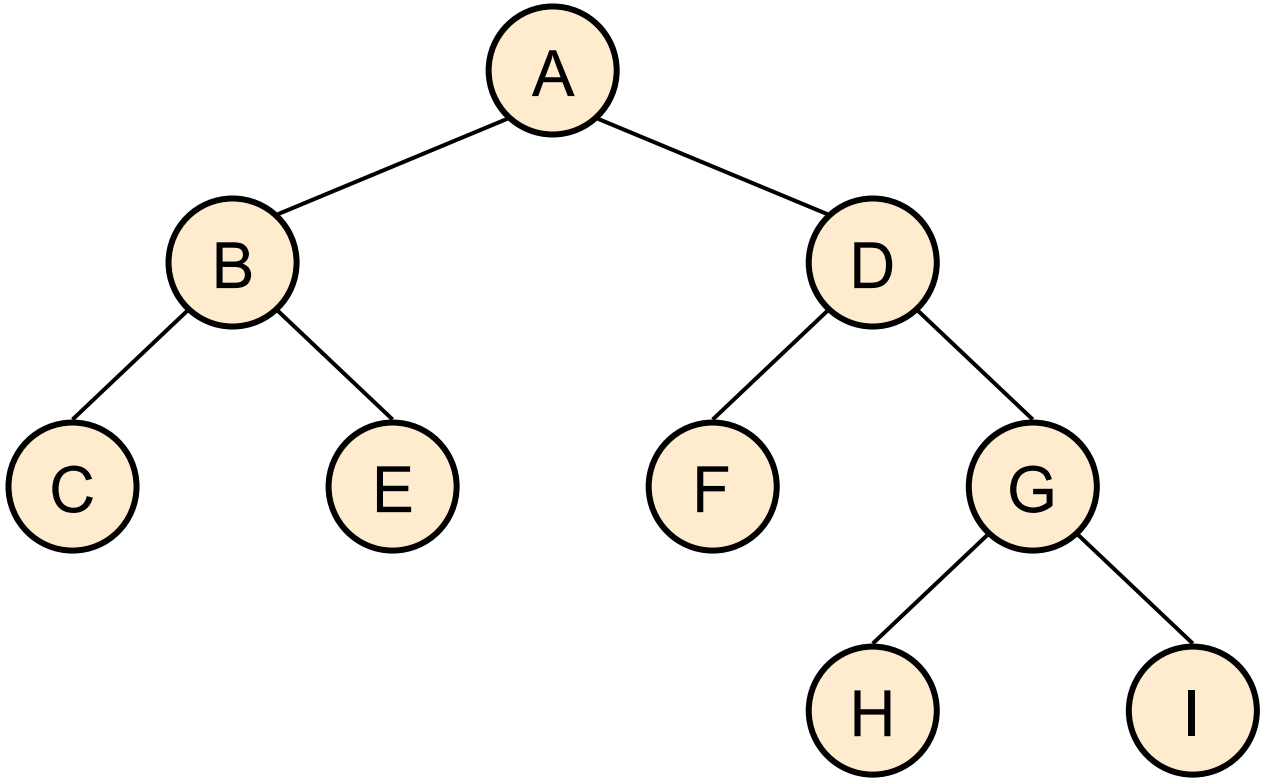
後順走査（帰りがけ順）

- postorderとも言う
- 子を調べ終わってから、自分のノードを調べる
- X→Y→A



演習 1

以下の木に対して、「幅優先探索」を行ったときのノードのたどる順と、「深さ優先探索（前順走査）」を行ったときのノードのたどる順をそれぞれ番号で示しなさい



本日の内容

木とは

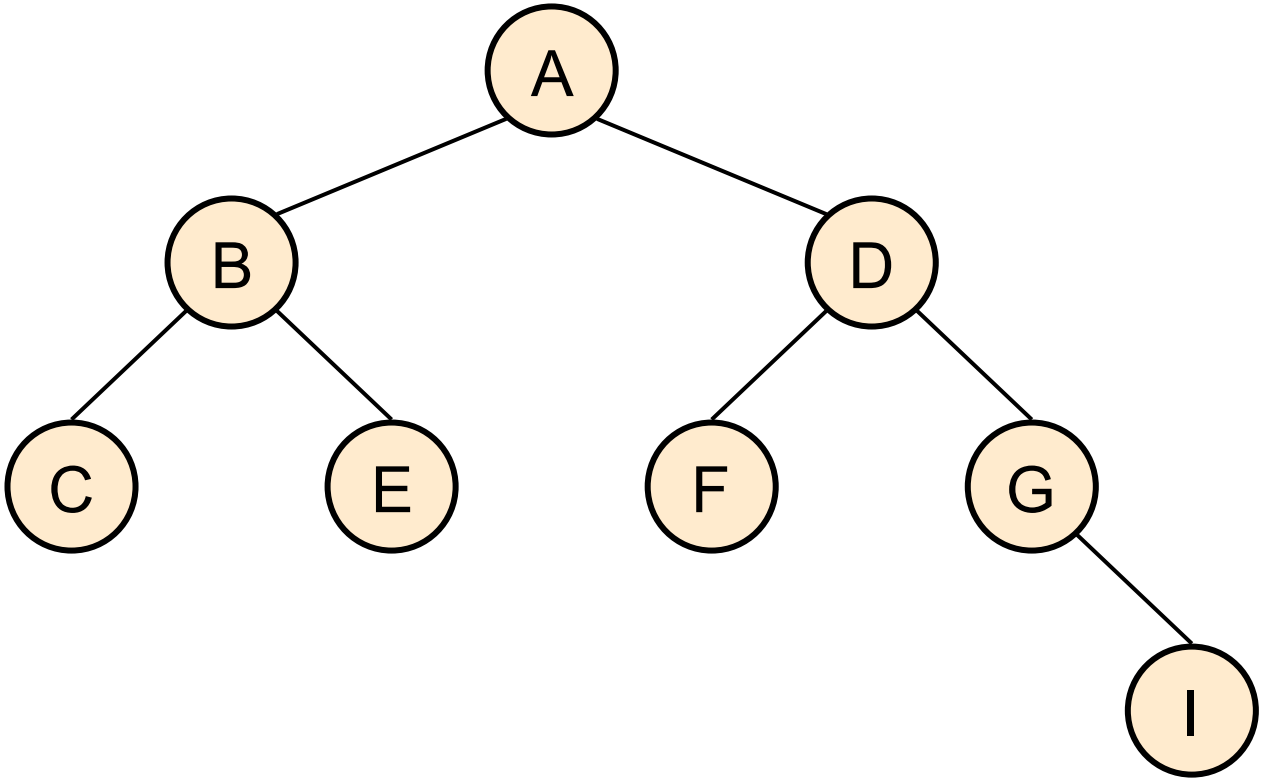
木の探索

2分木

2分探索木

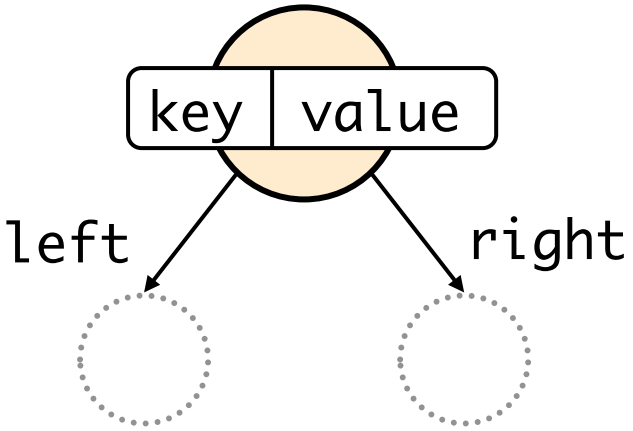
2分木

各頂点の子の数が最大2である木



2分木のノードを表すプログラム

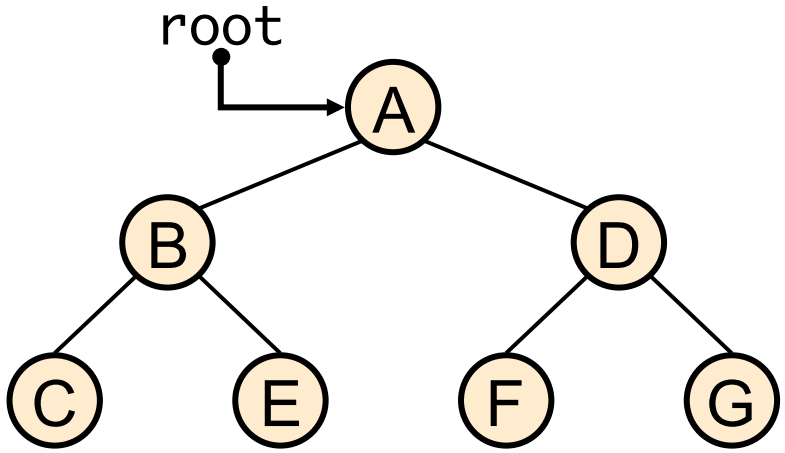
```
class BinNode {
    int key;
    String value;
    BinNode left, right;
}
```



- 左のノードと右のノードへの参照を持つ
- このプログラムでは、木にマップ（キーと値の組のデータ構造）を作成
  - 例: キーが氏名、値が電話番号
  - 例: キーが学籍番号、値が学生データ

木の使い方

- 根のノードを管理する変数を用意
- 根から目的のノードまでたどる
- 目的のノードまで道筋が分かっているなら O(log n) でアクセスできる

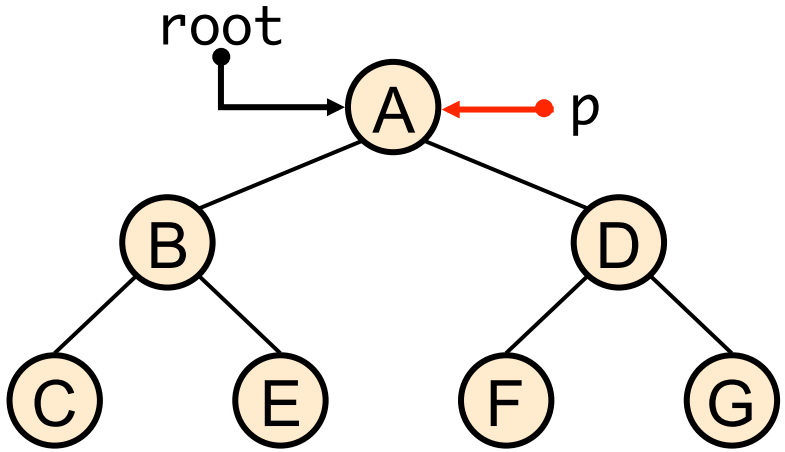


木の深さ優先探索（前順走査）

「ノード上の処理」 {
1. 自分のノードを調べる
   (ただし終点まで達したら戻る)
2. 左の子の「ノード上の処理」
3. 右の子の「ノード上の処理」

}

```
void preorder (BinNode p) {
    if (p == null) return;
    pに関する処理;
    preorder (p.left);
    preorder (p.right);
}
```



本日の内容

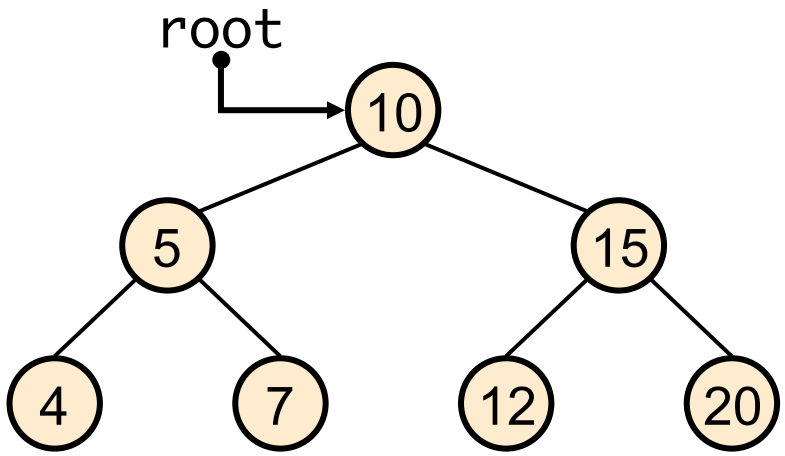
木とは
木の探索
2分木

2分探索木

2分探索木

各ノードから見て次を満たす木

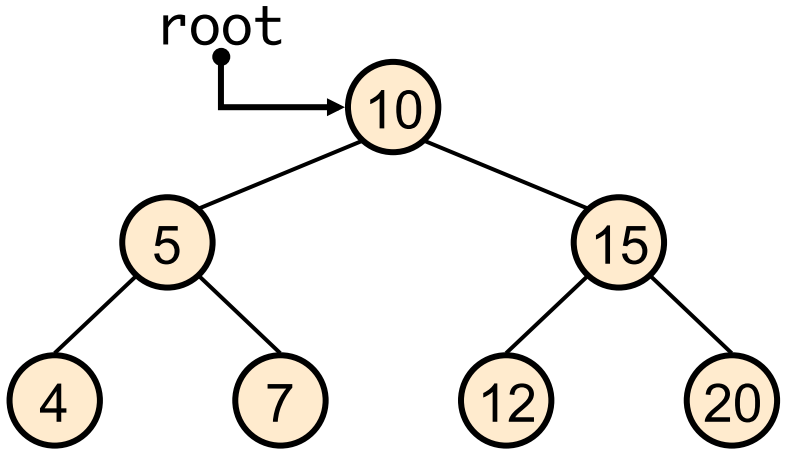
- **左**の子とその子孫のデータはすべて**小さい**
- **右**の子とその子孫のデータはすべて**大きい**



2分探索木の探索

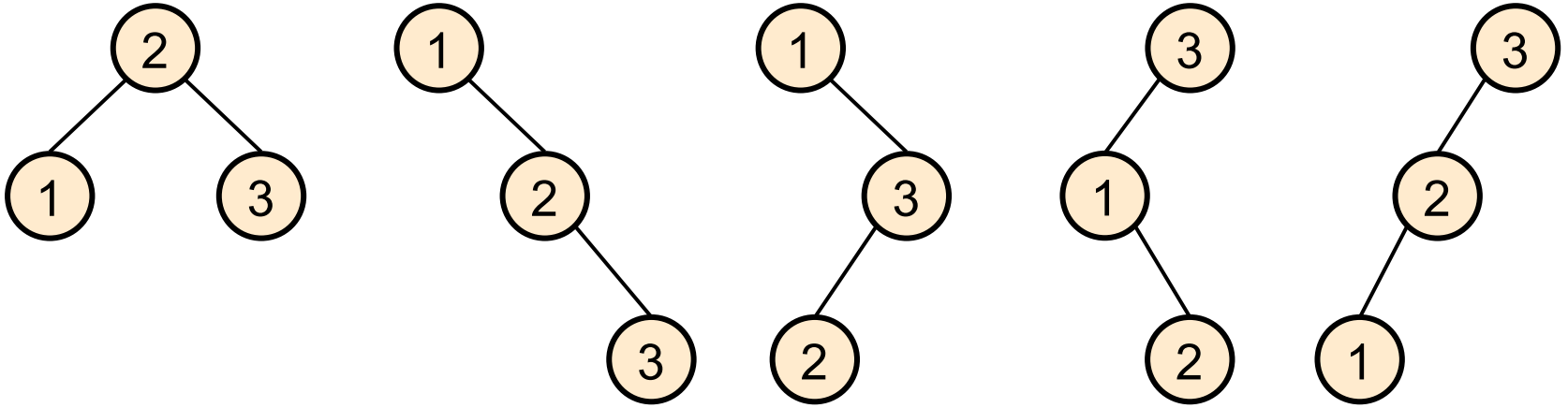
例: 「7」を探す

1. 「7」は「10」より小さい→左へたどる
2. 「7」は「5」より大きい→右へたどる
3. 「7」がみつかった



2分探索木の形と性能

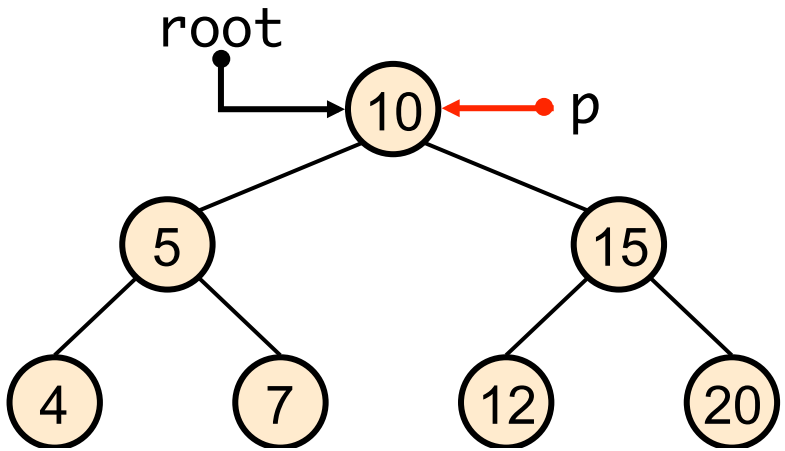
- データの集合を表す2分木の形は一とおりとは限らない
- バランスがとれ、木の高さが低い木ほど探索が高速
- 例: 1,2,3の3つの値からなる木の形



演習２　探索のプログラム

「ノード上の処理」 {
　見つけたい値と自分のノードのkeyを比較
　見つかった場合
　　自分のノードを探索結果として返す
　見つからなかった場合
　　見つけたい値が自分のノードのkey
　　より小さい場合、左の子に進む
　　みつけたい値が自分のノードのkey
　　より大きい場合、右の子に進む
　（ただし終点まで達したら見つからかったことになる）
}

```
BinNode search (BinNode p,int key){
    if (p == null) return null;
    if (p.key == key)
        ?
    else if (p.key > key)
        ?
    else
        ?
}
```

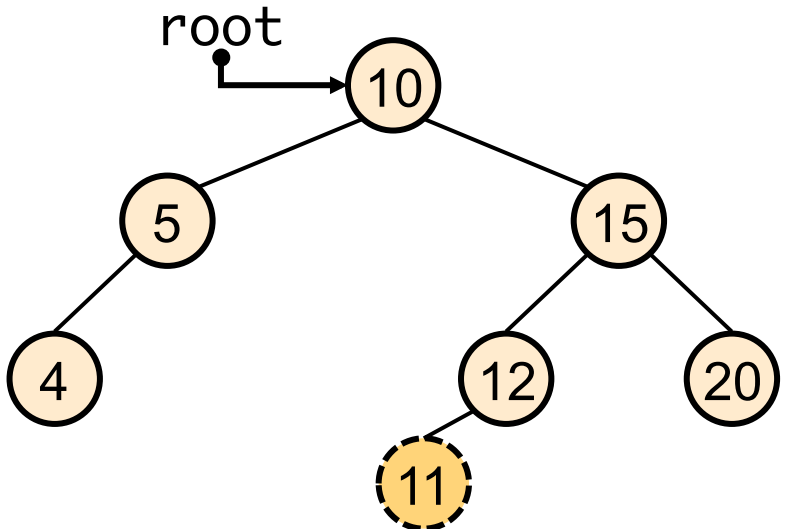


2分探索木へのデータの挿入

探索と同じ処理をし、行き止まりに達したらノードを追加

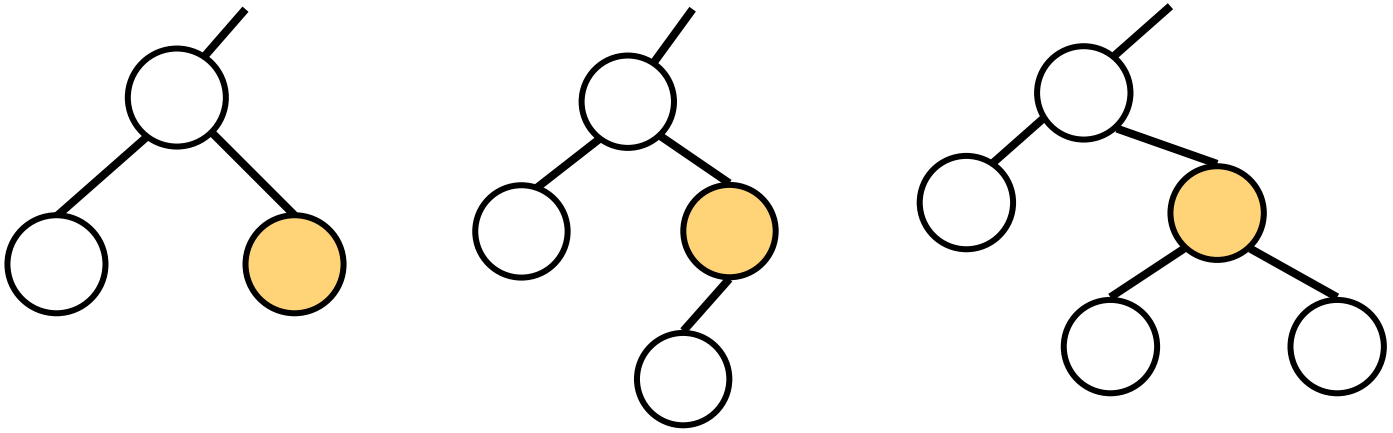
例: 「11」を加える

1. 「11」は「10」より大きい→右へたどる
2. 「11」は「15」より小さい→左へたどる
3. 「11」は「12」より小さい→左へたどる
   →先にはノードがない　ここに「11」を加える



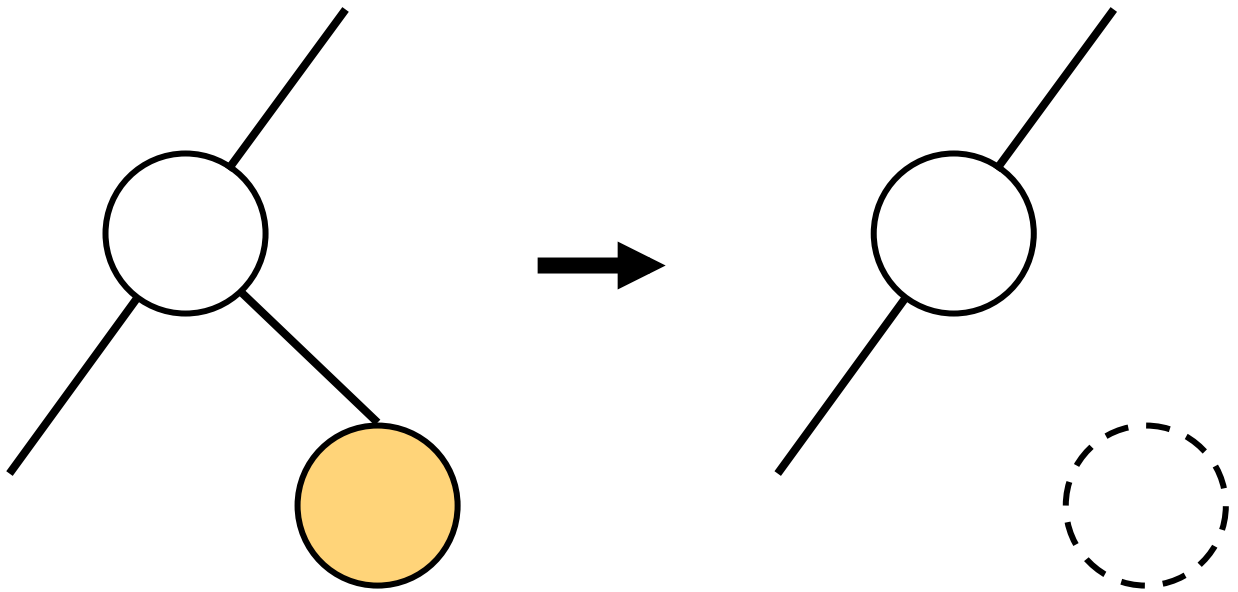
2分探索木からのデータ削除

- 削除ノードについて以下の３パターンに分けて考える
  - 葉の場合
  - 左右いずれか一方の子しかいない場合
  - 左右両方に子がいる場合



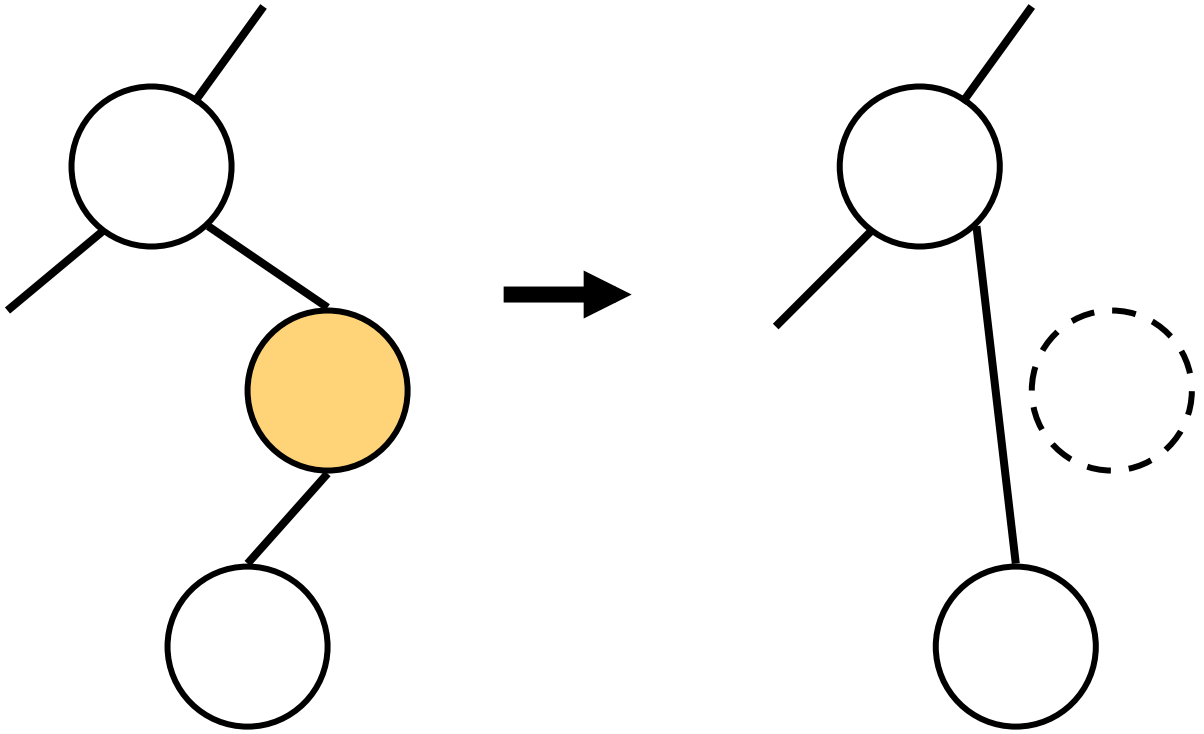
削除ノードが葉の場合

単にノードを取り除く（簡単）



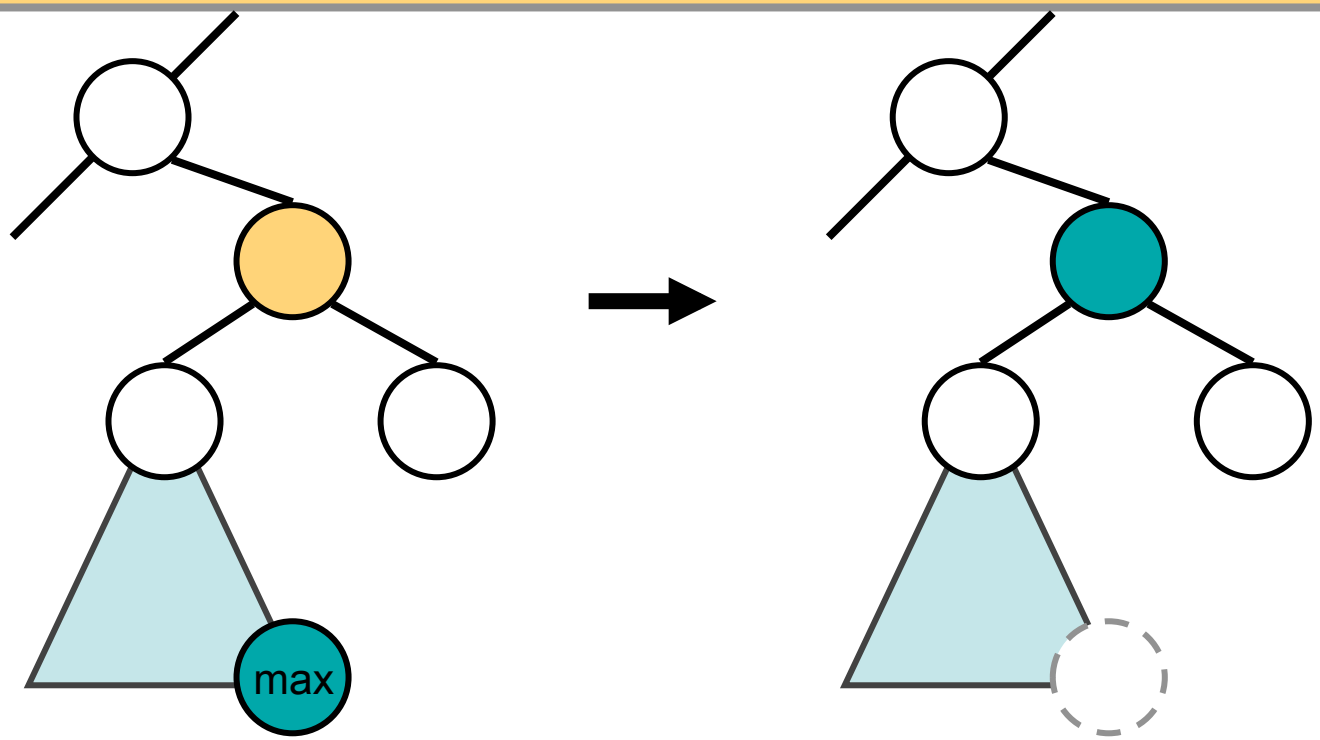
左右いずれか一方に子がいる場合

ノードを取り除き、ノードの子を代わりにする



左右両方に子供がいる場合

削除するノードからみて左の部分木の
最大のものをこのノードの代役にする
（右部分木の最小のものでも可）



演習３　削除処理を日本語で説明する

(rootが木の根のノードを指し、pが削除すべきノードを指すとする)

1．pが葉のとき

pが根であれば、 rootをnullにする。

そうでなければ、pの親がpの代わりにnullを指すようにする。

2．pの右だけに子がいるとき

pが根であれば、

そうでなければ、 ？

3．pの左だけに子がいるとき

pが根であれば、

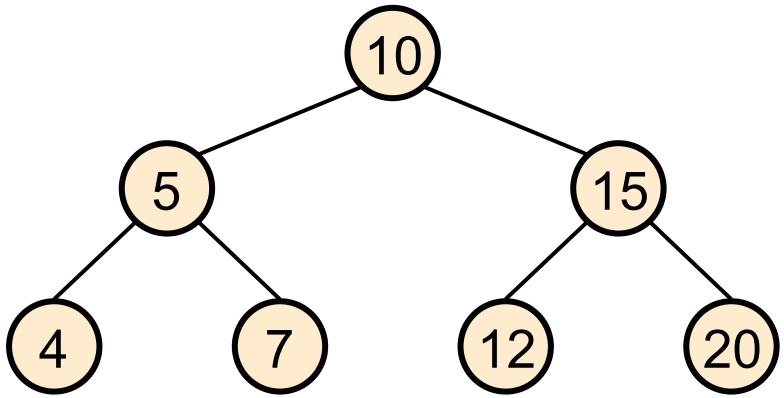
そうでなければ、 ？

4．1,2,3以外 （pには左右両方に子がいる）

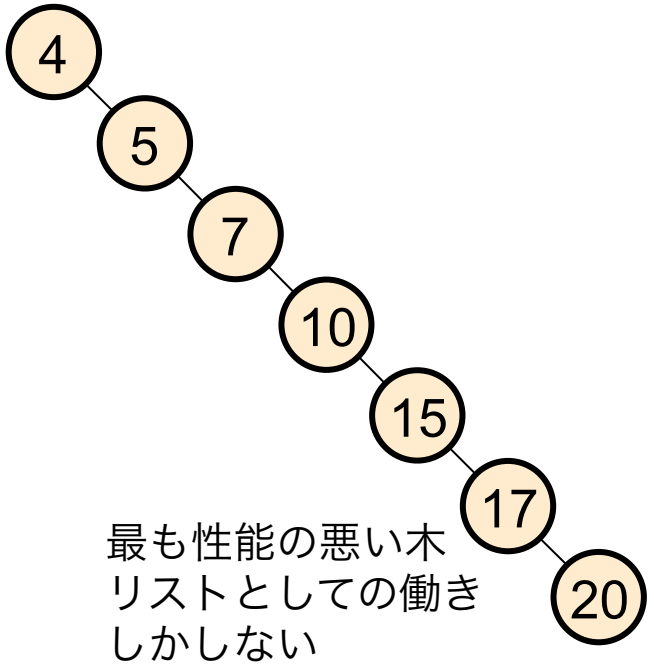
？

二分探索木の形と性能

- 探索の速度は目的のノードに到達するまでの比較回数で決まる
- バランスがとれ、木の高さが低い木ほど探索が高速



最も性能の良い木＝どのノードへも最短で到達できる
このように葉とその親ノード以外のすべてのノードの子の数が2である木を**平衡木**と呼ぶ



最も性能の悪い木
リストとしての働きしかしない